CS 4115 - SPINNER Language Reference Manual

William Beaver[1]
wmb2013@columbia.edu

Marc Eaddy
eaddy@cs.columbia.edu

Gaurav Khandpur
gaurav_k@cs.columbia.edu

Sangho Shin
ss2020@cs.columbia.edu

28 October 2003

[1]team leader

Contents

Chapter 1

Introduction

This Language Reference Manual (LRM) is organized as follows: this introduction outlines the document organization, briefly discusses the concurrency model required for the language as motivation for the language design, and highlights typographic conventions used. We then discusses the grammar hierarchy for SPINNER, proceeds to the lexical structure, discusses types values and variables, names, operators, blocks and statements, expressions, and assignment. Then, we describe the built-in functions to SPINNER and discuss the concurrency model in detail. In appendices, we provide the complete grammars, the beginnings of a function library for the language, and a few example programs.

SPINNER is a language designed for the expression of motion, its interaction with time in a physical system, and the issues of synchronicity that arise in such a system. (see the SPINNER whitepaper for discussion of motivation and target environment.) There are two primary functions needed in the target environment to express motion and time: the ability to seek a disc to a position within a specific duration and the ability to set the speed of a disc independent of setting the position. Upon these two primitives SPINNER adds control over system level settings like gain and volume. SPINNER also provides control over physical system settings that control the analogs of acceleration, deceleration, and motion/sound interaction. Features such as basic control flow (looping, branching, etc.), variables, access to primitive data structures, and access to disc level state information like current speed and position gives the SPINNER programmer levels of control that were previously unattainable. Additionally, SPINNER supports a concurrent processing model which allows process branching and interprocess communication that greatly eases the programming of multiple objects in tandem. (See Chapter 11 for more details.)

Throughout this document, references to one disc can be applied to `all` discs unless specified. The `typewriter font` is used for all keywords.

Chapter 2

Language construction

SPINNER is defined using Context Free Grammars (CFGs). These CFGs define the lexical conventions of the language, which are used to take the input (a series of ASCII characters), remove white space, and translate it into a sequence of tokens. The CFGs that define the syntactic grammar of SPINNER use the tokens defined by the lexical grammar to describe how the sequence of tokens form syntactically correct programs in the language.

Chapter 3

Lexical conventions

3.1 Line Terminator

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, not two.

```
NEWLINE: ('\n'|('\r' '\n') => '\r' '\n'|'\r')
        { $setType(Token.SKIP); newline();}
;
```

3.2 Tokens

Tokens in SPINNER define comments, identifiers, keywords, constants, literals, line terminators, operators, and separators in the language. White space is ignored in the system (as defined below) except where it is used to separate tokens.

3.3 Comments

SPINNER supports two types of comments. /* introduces a comment, which terminates with the characters */. // is also used to introduce a comment which terminates with the line separator (see below). Comments do not nest. /* and */ have no special meaning in comments that begin with //. // has no special meaning in comments that begin with /*.

```
CMT
        : ("/*" (options {greedy = false;} : NEWLINE
                 | ~( '\r' | '\n')
                )* "*/"
```

```
          | "//" (~( '\r' | '\n'))* NEWLINE
         )
        { $setType(Token.SKIP); }
;
```

3.4 White space

White space is defined as the ASCII space and horizontal tab characters, as well as line terminators.

```
WS : ( ' ' | '\t' )
{ $setType(Token.SKIP); } ;
```

3.5 Identifiers

An identifier is a sequence of letters and digits. First character must be a letter, and the underscore character counts as a letter. Case matters. Identifiers may have any length. Identifiers may not be a reserved keyword, a `boolean` literal, or the `null` literal.

```
protected LETTER : ('a'..'z' | 'A'..'Z') ;
: LETTER (LETTER | DIGIT | '_')* ;
```

3.6 Keywords

The following are reserved as keywords in the system and may not be used otherwise:

`all`	`double`	`repeat`
`boolean`	`else`	`return`
`break`	`for`	`then`
`while`	`func`	`void`
`do`	`if`	`not`
`and`	`or`	`until`

3.7 Literals

3.7.1 `double` literal

`double` literals consist of an integer part, a decimal part, and a fraction part. Integer and fraction parts each consist of a sequence of digits. The decimal part and fraction part are together optional.

```
protected DIGIT : '0'..'9' ;
ID options { testLiterals = true; }
: LETTER (LETTER | DIGIT | '_')* ;
NUMBER : (DIGIT)+ ('.' (DIGIT)+)?
       | '.' (DIGIT)+
;
```

3.7.2 `boolean` literal

The `boolean` type has two values, represented by the literals `true` and `false`, formed from ASCII letters.

```
boolean:
      ["true" | "false"]
```

3.7.3 `null` literal

The `null` type has one value, the `null` reference, represented by the literal `null`, which is formed from ASCII characters. A `null` literal is always of the `null` type.

```
null:
     "null"
```

3.7.4 String literal

A string literal is a sequence of characters surrounded by double quotes. String literals do not contain newline or double quotes.

```
STRING : '"'! ('"' '"'! | ~('"'))* '"'! ;
```

3.8 Separators

The following nine ASCII characters are the separators (punctuators):

```
(   )   {   }   [   ]   ;   ,   .
```

3.9 Operators

The following seventeen objects represent operators:

```
<  >  <=  >=  =  not
+  -  *  /  %
++  --  and  or
:=  ||
```

Chapter 4

Types, values and variables

4.1 Introduction

SPINNER is a strongly typed language; which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time.

4.2 Data types

SPINNER supports the `double`, `boolean`, and array data types.

4.2.1 `double`

`double` variables consist of an integer part, a decimal part, and a fraction part. Integer and fraction parts each consist of a sequence of digits. The decimal part and fraction part are together optional. A `double` can be optionally signed.

```
protected DIGIT : ’0’..’9’ ;
ID options { testLiterals = true; }
: LETTER (LETTER | DIGIT | ’_’)* ;
NUMBER : (DIGIT)+ (’.’ (DIGIT)+)?
       | ’.’ (DIGIT)+
;
```

4.2.2 `boolean`

The `boolean` type has two values, represented by the literals `true` and `false`, formed from ASCII letters.

`boolean` → [TRUE | FALSE]

4.2.3 Array

A SPINNER array can contain any of the data types supported, including other arrays. Arrays are indexed $1 \ldots N$ where N is the size of the array. Declaration of an array type later allows programmer to access the getSize() function which returns the number of elements in the referenced array. SPINNER array elements are referenced by array name[index] where the index refers to an element of the array. If index does not refer to a valid element in the array (i.e. the index is not within $1 \ldots N$) a compiler error occurs.

4.3 Variables

A variable is a storage location and has an associated type. A variable always contains a value that is assignment compatible with its type.

4.3.1 Naming

Variables definitions consist of a type declaration and variable name and an optional initialization. SPINNER does not support multiple variable declarations of the same type in the same declaration; each variable must be individually declared. SPINNER does not support duplicate variable name declarations within the same scope and will throw an error at compile time.

Example:

```
type varName;

type varName := value;
```

4.3.2 Scope

Global

SPINNERs global scope is available to all procedures, functions and scopes and is defined as everything within the topmost structure of the program.

Local

SPINNER provides local scoping. Local scope is maintained for each procedure, function and block statement and propagates to their children process/scopes. Local scope is the present scope plus child scopes.

4.3.3 Initial values

`boolean` values are initialized to `false` if no value is provided at declaration.
`double`s are initialized to 0.0 at instantiation if no value is provided.
Array elements are initialized to `null` at declaration.

Chapter 5

Conversion and promotion

5.1 `boolean` → `double`

`boolean` values are converted as follows: `true` → 1 and `false` → 0 when used in places where type `double` is expected.

5.2 `double` → `boolean`

`double` values are converted to `boolean` values as follows: $-\infty \ldots 0 \rightarrow$ `false` and $1 \ldots +\infty \rightarrow$ `true`.

5.3 `double` truncation

`double` are used in situations where a traditional integer values are required (as array indices, disc number references, etc.). When this occurs, SPINNER truncates the value by dropping the decimal and fraction part of the number if it was provided. If only the decimal and fractional part are provided, SPINNER interprets this value as 0 (zero).

5.4 `null`

`null` value is not converted and will throw error at compile time if used when expecting a declarable type.

Chapter 6

Names

Names are used to refer to entities declared in a program. A declared entity is a parameter, a local variable, or a function. Names in programs are simple, consisting of a single identifier. Every declaration that introduces a name has a scope, which is the part of the program text within which the declared entity can be referred to by a simple name. The name of a field, parameter, or local variable may be used as an expression. The name of a function may appear in an expression.

6.1 Declarations

A declaration introduces an entity into a program and includes an identifier that can be used in a name to refer to this entity. A declared entity is one of the following:

- A function
- A parameter
- A local variable, one of the following:
  - A local variable declared in a block
  - A local variable declared in a `for` statement

6.2 Scope of a declaration

The scope of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name (provided it is visible). A declaration is said to be in scope at a particular point in a program if and only if the declaration's scope includes that point.

The scoping rules for various constructs are given here:

1. The scope of a type imported by a import declaration is the same as if the declarations occurred within the program itself. For example, a variable which was global to the program being imported is global in the program it was imported into as import statements can only occur within the global scope.
2. The scope of a parameter of a function is the entire body of the function.
3. The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement.
4. The scope of a local variable declared in the initialization part of a `for` statement includes all of the following:
    - Its own initializer
    - Any further declarators to the right in the initialization part of the `for` statement
    - The Expression and update parts of the `for` statement
    - The contained Statement

6.3 Naming conventions

6.3.1 Function names

Function names should be verbs or verb phrases, in mixed case, with the first letter lowercase and the first letter of any subsequent words capitalized. Here are some additional specific conventions for function names:

- Functions to get and set an attribute that might be thought of as a variable V should be named getV and setV.
- A function that returns the length of something should be named size, as in arrays.
- A function that tests a boolean condition V about an object should be named isV.

6.3.2 Constant names

Note: SPINNER does not support constants and as such will not protect values of variables as you would constants. By naming variable according to a standard they can be easily recognized by programmers and appropriate care taken to protect their the integrity of their values.

The names of variables to be treated as constants should be a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech. Examples of names for constants include MIN_VALUE, MAX_VALUE, MIN_RADIX, and MAX_RADIX.

6.3.3 Local variable and parameter names

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words. Of course, variable and parameter names cannot be keywords.

Chapter 7

Blocks and statements

7.1 Blocks

A block is a sequence of statements, local class declarations and local variable declaration statements within braces.

```
main_statement
        :outer_statement
        |func_definition
;

outer_statement
        :statement
        |declaration SEMI
        |parallel_statement
;

statement
        :if_statement
        |for_statement
        |while_statement
        |repeat_statement
        |assignment SEMI!
        |func_call SEMI!
        |create_disk SEMI!
        |SEMI!
;
```

```
inner_statement
        :statement
        |break_statement
        |LBRACE! (main_statement)* RBRACE!
;
```

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

7.2 Local variable declaration statements

```
variable_declaration
        : type (ID | assignment_with_decl | array_decl)
;

type
        : "double"
        | "boolean"
        | "void"
;
```

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a `for` statement. In this case it is executed in the same manner as if it were part of a local variable declaration statement.

7.2.1 Local variable declarators and types

Each declarator in a local variable declaration declares one local variable, whose name is the Identifier that appears in the declarator. Each declarator in a local variable declaration declares one local variable, whose name is the Identifier that appears in the declarator.

Examples:

```
type varName;  \\variable without initialization

type varName = value;  \\variable with initialization

type varName[size];  \\array declaration
```

7.2.2 Scope of local variable declarations

The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement. The name of a local variable v may not be redeclared as a local variable of the directly enclosing function, block or initializer block within the scope of v, or a compile-time error occurs.

The scope of a local variable declared in a `for` statement is the rest of the `for` statement, including its own initializer.

7.2.3 Execution of local variable declarations

A local variable declaration statement is an executable statement. Every time it is executed, the declarators are processed in order from left to right. If a declarator has an initialization expression, the expression is evaluated and its value is assigned to the variable. If a declarator does not have an initialization expression, then a SPINNER initializes the variables as defined in section 4.3.3.

Each initialization (except the first) is executed only if the evaluation of the preceding initialization expression completes normally. Execution of the local variable declaration completes normally only if evaluation of the last initialization expression completes normally; if the local variable declaration contains no initialization expressions, then executing it always completes normally.

7.3 Statements

There are many kinds of statements in the SPINNER programming language. Most correspond to statements in the Java, C, and C++ languages. As in C and C++, the `if` statement of the Java programming language suffers from the so-called "dangling else problem". SPINNER, like the Java programming language, C, and C++ and many programming languages before them, arbitrarily decree that an `else` clause belongs to the innermost `if` to which it might possibly belong. This rule is captured by the following grammar:

```
statement
        :if_statement
        |for_statement
        |while_statement
        |repeat_statement
        |assignment SEMI!
        |func_call SEMI!
        |create_disk SEMI!
        |SEMI!
;
```

```
inner_statement
        :statement
        |break_statement
        |LBRACE! (main_statement)* RBRACE!
;
```

7.3.1 Empty statement

An empty statement does nothing.

```
EmptyStatement:
        ;
```

Execution of an empty statement always completes normally.

7.3.2 Expression statements

Certain kinds of expressions may be used as statements by following them with semicolons. An expression statement is executed by evaluating the expression; if the expression has a value, the value is discarded. Execution of the expression statement completes normally if and only if evaluation of the expression completes normally. SPINNER allows only certain forms of expressions to be used as expression statements. (see Chapter 8)

7.3.3 `if then` | `if then else`

In both forms of the `if` statement, the expression is evaluated, and if it evaluates equal to `true`, the first substatement is executed. In the second form, the second substatement is executed if the expression is `false`.

```
if_statement
        : "if" LPAREN! expr RPAREN! "then"! inner_statement
        (options {greedy=true;} : "else"! inner_statement)?
;
```

7.4 Iteration

7.4.1 `while...do`

The substatement is executed repeatedly so long as the value of the expression remains equal `true`. The test preceded one iteration of the statement.

```
while_statement
        : "while" LPAREN! expr RPAREN! "do" inner_statement
;
```

7.4.2 `repeat`...until

The substatement is executed repeatedly so long as the value of the expression remains equal `true`. The test follows after at least one iteration of the statement.

```
repeat_statement
        : "repeat" inner_statement "until" LPAREN! expr RPAREN!
;
```

7.4.3 `for`

The first statement is evaluated once, establishing the initialization of the loop. There is no restriction on its type. The second is an expression; it is evaluated before each iteration and if it becomes `false`, the `for` statement is terminated. The third expression is evaluated after each iteration and thus defined the reinitialization of the loop.

```
for_statement
        : "for" LPAREN! (for_init)? SEMI! (expr)? SEMI! (for_increment)? RPAREN! inner_statemen
;

for_init
        :assignment
        |declaration
;

for_increment
        :assignment
;
```

7.4.4 `break` statement

`break` statement may appear in any blockstatement (except the global statement) and it terminates the execution of the smallest enclosing statement and returns control to the statement following the terminated statement.

```
break_statement
        : "break" SEMI!
;
```

7.4.5 `return` statement

A `return` statement returns control to the invoker of a function.

```
return_stmt
        : "return" (expr)?
;
```

A `return` statement with no Expression must be contained in the body of a function that is declared, using the keyword `void`, not to return any value. A `return` statement with no Expression transfers control to the invoker of the function that contains it. A `return` statement with no Expression always completes abruptly, the reason being a `return` with no value.

A `return` statement with an Expression must be contained in a function declaration that is declared to return a value or a compile-time error occurs. The Expression must denote a variable or value of some type T, or a compile-time error occurs. The type T must be assignable to the declared result type of the function, or a compile-time error occurs.

A `return` statement with an Expression transfers control to the invoker of the function that contains it; the value of the Expression becomes the value of the function invocation. More precisely, execution of such a `return` statement first evaluates the Expression. If the evaluation of the Expression completes abruptly for some reason, then the `return` statement completes abruptly for that reason. If evaluation of the Expression completes normally, producing a value V, then the `return` statement completes abruptly, the reason being a `return` with value V.

It can be seen, then, that a `return` statement always completes abruptly.

7.5 Concurrency operator ||

Concurrent execution paths are created by setting block statements off and joining them with ||. When a block containing thread is encountered, SPINNER forks and concurrently processes each block. The entire block completes when all threads terminate. || binds tightly to adjoining block statements and block statements *only*. For example

```
{blockstatement} || {<blockstatement}
```

is a valid concurrency operator.

```
{blockstatement} || setSpeed(i, 4, 4000);
```

is an error (parallel block with standalone function) as is the following:

```
 function; || setSpeed(i, 4, 4000);
```

(a standalone function parallel to another standalone function)

This is valid:

```
{function} || {setSpeed(i, 4, 4000); }
```

because each function is enclosed as a blockstatment.

Termination of a series of concurrent statements occurs when all statements have completed execution. Until all the processes complete or are terminated, program flow will halt. Where possible, the compiler will indicate deadlock situations, but care should be taken by the programmer to avoid such situations.

Concurrency can be nested as long as proper block statement nesting syntax is used. When nested, control lives in the bottommost processes of the process fork tree and flows upwards.

For a full description of SPINNER's concurrency model, see Chapter 11.

Chapter 8

Expressions

When an expression in a program is evaluated (executed), the result denotes one of three things:

1. A variable (in C, this would be called an lvalue)
2. A value
3. Nothing (the expression is said to be `void`)

Evaluation of an expression can also produce side effects, because expressions may contain embedded assignments, increment operators, decrement operators, and method invocations.

An expression denotes nothing if and only if it is a function call that invokes a function that does not return a value, that is, a function declared `void`. Such an expression can be used only as an expression statement, because every other context in which an expression can appear requires the expression to denote something. An expression statement that is a functional call may also invoke a function that produces a result; in this case the value returned by the function is quietly discarded.

Value set conversion is applied to the result of every expression that produces a value.

8.1 Evaluation order

In SPINNER, operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated.

Every operand of an operator (except the conditional operators `and` `or`) appears to be fully evaluated before any part of the operation itself is performed.

Evaluation respects parentheses and precedence.

Argument lists are evaluated left-to-right

8.2 Precedence

SPINNER has the following precedence levels (from highest to lowest):

```
or and
*  /  mod(%)
+  -  ++  --
< > <= >= =
:=
```

8.3 Array creation

An array instance creation expression is used to create new arrays.

```
declaration:
            type ID[expr];
```

An array creation expression creates an object that is a new array whose elements are of the type specified by the type.

The type of each dimension expression within a expr must be an integral type, or a compile-time error occurs. Each expression undergoes promotion which is double truncation of the decimal part, or a compile-time error occurs. SPINNER does not support array intialization at declaration. If an array initializer is provided, an error is generated at compile time.

8.3.1 Array access expressions

An array access expression refers to a variable that is a component of an array.

```
access:
       ID[expr]
```

An array access expression contains two parts, the array reference ID (before the left bracket) and the index expression (within the brackets). Note that the array reference ID may be a name or any primary expression that is not an array creation expression. The type of the array reference expression must be an array type (call it T[], an array whose components are of type T) or a compile-time error results. Then the type of the array access expression is T.

The index expression undergoes unary numeric promotion. If the index type is `double`, the decimal part is truncated (unofficially promoted to a integer.)

The result of an array reference is a variable of type T, namely the variable within the array selected by the value of the index expression.

An array access expression is evaluated using the following procedure:

- The array reference expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason and the index expression is not evaluated.
- Otherwise, the index expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason.
- Otherwise, if the value of the array reference expression is `null`, then a NullPointer error is raised.
- Otherwise, the value of the array reference expression indeed refers to an array. If the value of the index expression is less than zero, or greater than or equal to the array's length, then an ArrayIndexOutOfBounds error is raised.
- Otherwise, the result of the array access is the variable of type T, within the array, selected by the value of the index expression.

8.3.2 Array assignment expression

Array assignment expressions come in two flavors. One uses explicit indexing, the other implicit indexing. To assign values to an array explcitly, each element of the array is indexed individually and a value assigned. For example:

```
array[1]:= 3.0; //set the first element of array to 3.0

array[i]:= k; //set the ith element of array to value stored in variable k
```

If the array index is out of bounds, a compile error will occur. If the assignment types are not consistent, SPINNER attempts implcit type conversion (Chapter 5). If this fails, a compile time error will occur.

For implicit array assignment, all or some of the array elements are referenced and assigned. For example:

```
array[]:= {3.0, 4.0}; //set the first and second elements of array to 3.0 and 4.0
```

The assignment begins from the lowest (leftmost) index and places the values in the bracketed statement accordingly. If the number of elements be assigned exceeds the number of elements in the array, a compile error occurs.

8.4 Function invocation expressions

A function invocation expression is used to invoke a function.

```
func_call
        : ID LPAREN (params)? RPAREN
;
```

8.5 Postfix expressions

8.5.1 Postfix increment (x++)

A unary expression proceeded by a ++ operator is a postfix increment expression. The result of the unary expression must be a variable of a numeric type. Postfix increment in SPINNER means the value of the variable being incremented is incremented after the variable is used.

8.5.2 Postfix decrement (x−−)

A unary expression proceeded by a −− operator is a postfix decrement expression. The result of the unary expression must be a variable of a numeric type. Postfix decrement in SPINNER means the value of the variable being decremented is decremented after the variable is used.

8.6 Unary operators

SPINNER only support one unary operator: logical negation

8.6.1 Logical negation operator `not`

An expression prefixed with the logical negation operator will be evaluated and then the result negated its result.

8.7 Multiplicative operators

8.7.1 Multiplication operator *

The binary * operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects.

8.7.2 Divide operator /

The binary / operator performs division, producing the quotient of its operands. The left-hand operand is the dividend and the right-hand operand is the divisor.

8.7.3 Remainder operator %

The binary % operator is said to yield the remainder of its operands from an implied division; the left-hand operand is the dividend and the right-hand operand is the divisor. In C and C++, the remainder operator accepts only integral operands, but in the SPINNER programming language, it also accepts floating-point operands.

8.8 Additive operators

8.8.1 Addition operator +

The binary + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. Addition is a commutative. Addition is associative.

8.8.2 Subtraction operator -

The binary - operator performs subtraction, producing the difference of two numeric operands.

8.9 Relational operators

SPINNER supports the following relational operators:

```
>  <   <=  >=  =
```

The type of a relational expression is always `boolean`:

- The value produced by the $<$ operator is `true` if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the $\leq$ operator is `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the $>$ operator is `true` if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the $\geq$ operator is `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the $=$ operator is `true` if the value of the left-hand operand is equal to the value of the right-hand operand, and otherwise is `false`.

8.10 Logical operators

8.10.1 Logical `and`

For `and`, the result value is `true` if both operand values are `true`; otherwise, the result is `false`. The `and` operator is lazy; meaning that if the first operand is `false`, the statement will immediately evaluate to `false` without evaluation of the second operand. It is syntactically left-associative (it groups left-to-right).

8.10.2 Logical `or`

The result value is `true` if either of the operand values are `true`. The `or` operator is lazy; meaning that if the first operand is `true`, the statement will immediately evaluate to `true` without evaluation of the second operand. It is syntactically left-associative (it groups left-to-right).

8.11 Assignment operator :=

Assignment operator := places the value of the right operand into the left operand. Type conversion (as explained in Chapter 5) attempts to fix assignments. If the assignment fails, a compile time error will occur.

Chapter 9

Functions

9.1 Declaration and format

A function in SPINNER is named block of statements that accepts an optional parameter list of input and returns an optional value using the `return` statement (section 7.4.5).

Functions are declared by first identifying its return type. If a function is not going to return anything, it should declare type `void`. Then the keyword `func` is used to identify that this is a function declaration followed by the name of the function. Finally, a list of parameters are declared for the function. These parameters become variables accessible within the local scope of the function. After the function declaration, a block statement contains the statements and expressions of the function.

```
func_definition
        : type "func" ID LPAREN! (args)? RPAREN! LBRACE! func_body RBRACE!
;

func_body
        : (main_statement)*
;

args
        : arg (COMMA! arg)*
;

arg
        : type ID
;
```

```
params
        : expr (COMMA! expr)*
;
```

Every function must terminate abruptly through the use of the `return` statement. If a function declares a return type of `void`, then a `return` statement with no Expression must be called to terminate the function. If a function declares a return type of some SPINNER data type, then the `return` statement must be called with an Expression than matches the declared return type. If the Expression returns a type that is inconsistent with the declared return type, SPINNER will attempt to promote the return value using its promotion and conversion rules (Chapter 5). If promotion or conversion is not possible, the function will throw an error.

Chapter 10

Built-in SPINNER functions

SPINNER contains a number of function which are native to the language. These are provided here as a reference. Each function can be called as a standalone expression statement or used in conjunction with other statements. For each, we define the return type, name, parameter list, process description and some examples.

10.1 seek

command	seek
return type	`void`
syntax	seek([discnum | `all`], SeekPosition, Time);

parameters:

[*discnum* | `all`] is disc number (truncated `double`) that this command applies to. `all` is a valid disc reference for all the discs.

SeekPosition is the absolute numerical position (`double`) that the disc should seek to. 0 is home position, 1.0 is one revolution from home, 2.0 is two revolutions from home, etc. Negative values for SeekPosition indicate counter clockwise rotation.

Time is the number of milliseconds (truncated `double`) the disc has to get to SeekPosition.

The seek command specifies the position a disc should seek from it's present position and the timeframe in which it has to do it.

The seek command is synchronous. It effectively sets a disc's velocity and then waits for TimeN milliseconds. After completion of line command, the disc will stop.

Example

```
//disc 1, spin one clockwise rotation to position 1 in 1000ms=1s and then stop
seek(1,1,1000);
```

Seek commands issued with a wait break between them will behave as follows: the first issuance of the command establishes the position to seek to and the time in which to do it. A wait(N) command allows this motion to proceed for N ms, then the next line command is issued and the disc immediately executes this commands. Disc position and speed when command two is issued are relative to the state after executing command one for N ms. Seek (and setSpeed) commands issued sequentially without intervening wait statements result in the execution of the last command.

10.2 setSpeed

command	setSpeed
return type	`void`
syntax	setSpeed([DiscNum | `all`], TargetSpeed, RampTime);

parameters:

DiscNum is the disc number that this command applies to. `all` is a valid disc reference.
TargetSpeed is the target speed in revolutions per second (`double`).
RampTime is the number of milliseconds (`double`) you have to achieve the TargetSpeed.

setSpeed is an acceleration function. It tells a disc to spin at TargetSpeed revolutions per second and to get to the TargetSpeed in RampTime seconds. Acceleration to TargetSpeed is linear. setSpeed commands are absolute; execution of the command replaces the motion established by a previous setSpeed (or seek)command.

Examples:

```
//disc 1 spin at .75 revs/s and take 10 secs to get there.
setSpeed(1, .75, 10000);

//all discs spin at -.5 revs/ sec [.5rev/sec ccw] and take 5 secs to get there
setSpeed(all, -.5,  5000);

//all discs spin at 2 rev/sec and take 4.5 secs to get there
setSpeed(all, 2.0, 45000);
```

10.3 getSpeed

getSpeed(discnum) returns the `double` value of the reference disc's speed at the time the function is called. If the disc is spinning counter clock-wise, the speed returned will be negative.

10.4 getPosition

getPosition(discnum) returns the `double` value of the disc's position at the time the function is called. Position is reported as number of revolutions away from the zero (0) position. Counter clock-wise revolutions report a negative value.

10.5 wait

Wait(time) takes an integer constant or variable and returns `void`. It instructs the system to pause the current branch for the time indicated (in milliseconds). The time value is truncated if `double` is provided.

10.6 waitUntil

waitUntil(condition, timeout)

The waitUntil function takes a `boolean` expression argument and a timeout value. Its purpose is to block the currently executing branch until the `boolean` expression evaluates to `true` or the timeout expires. This is useful for waiting until a certain condition occurs, for example, waiting for the angle of disc 1 to be 30 degrees, or for synchronizing the actions of two or more discs.

You can specify a timeout value: $= \ -1$ (Infinity)
$\geq \ 0$ (Number of milliseconds to wait before timing out)

1. When the boolean expression evaluates to `true`, waitUntil stops blocking and returns the number of milliseconds spent blocked as an integer. The value returned will never be greater than the timeout value unless timeout is -1 (infinity).

2. The simulator will attempt to evaluate the waitUntil condition.

   - If `true`, the instruction ptr for that branch is incremented.
   - Otherwise, it is not incremented. This means that we will continue evaluating the waitUntil conditional until it becomes `true`, possibly forever.

3. While one branch is "stuck" until the condition is `true`, other branches continue executing as normal. When waitUntil evaluates to `true`, the simulator will output a SPINscript wait statement for the time it took the condition to be evaluated to `true`. (Actually, the wait statement may be interleaved with other branch output.)

example:

```
{
   seek(1, 10, 10000);
   waitUntil(getAngle(2) = 180, -1);
   setSpeed(1, getSpeed(2));
}
||
{
   seek(1, 10, 5000);
   wait(10000);
}
```

10.7 setGain

command	setGain
return type	`void`
syntax	setGain([DiscNum | `all` | “master”], val) ;
parameters:	

[*DiscNum* | ***`all`*** | *“master”*] is the disc number that this command applies to. `all` and “master” are a valid disc references.

val is a continuous value (`double`) on (0,1) to set gain.

setGain sets the gain for the appropriate object (disc or master). setGain returns no value.

Examples

```
setGain(1, 1.);  //set disc 1 gain to 1.0

setGain(all, 0.25); //set all disc's gain to .25

setGain(master, 0.22); //set system gain to .22
```

10.8 getGain

syntax getGain(discnum | master]); getGain returns a `double` indicating the gain value for the requested object or `null` if the gain value is unknown.

10.9 setFile

setFile([discnum | `all`], ”filename”); setFile indicates the sound file for the appropriate object (disc or master). setFile returns no value.

10.10 isFileSet

isFileSet(discnum); isFileSet returns a `boolean` indicating whether a file has been set for a particular object. `true` indicates that a file has been set. `false` indicates a file has not been set. `all` is NOT a valid parameter for discnum.

10.11 setReverb

command	setReverb
return type	`void`
syntax	setReverb([“send” | “return”], val);

parameters:

[*“send”* | *“return”*] is string literal used to set appropriate sub feature of reverb. required.
val is continuous value (`double`) to set reverb to. required.

setReverb sets the system reverb. setReverb returns no value.
Examples

```
setReverb(‘‘send’’, 1.); //set send reverb to 1.0

setReverb(‘‘return’’, 0.25); //set return reverb to .25)
```

10.12 getReverb

getReverb([send | return]);

Get reverb returns the reverb value of the appropriate sub feature of reverb. If the sub feature was not set, getReverb returns `null`. [*send* | *return*] are literals that refer to the specific sub feature of getReverb.

10.13 setSpinMult

command	setSpinMult
return type	`void`
syntax	setSpinMult([discnum | `all`], val);

parameters:

[*discnum* | *`all`*] is the disc number that this command applies to. `all` is a valid disc references.
val is a value (`double`) to set the spinMult to.

The sound and the rotation in SPIN are locked together, and spinmult is like the gear ratio between them. When spinmult is 1.0, then the sound sample will play back at its original speed when the wheel is turning at 1.0 RPS. When the wheel turns at 2.0 RPS, the sample will play back at double speed and at a higher pitch (1 octave higher, which is double pitch). At .5 RPS, the sound would play back at half speed/half pitch. at -1.0 RPS, the sound will play back at normal speed, but backwards. When spinmult is .5, then the sound sample will play back at 1/2 speed and 1/2 pitch when the wheel is turning at 1.0 RPS, etc.

10.14 getSpinMult

getSpinMult(discnum);

for a referenced object, getSpinMult returns the `double` value of the spinMult. If spinMult value is unknown, getSpinMult returns `null`.

10.15 setSampleStart

command	setSampleStart
return type	`void`
syntax	setSampleStart([discnum | `all`], val);

parameters:

[*discnum* | *`all`*] is disc number that this command applies to. `all` is a valid disc reference.
val is the point in time within the defined sound file that you would like the sample to start.

see setLength (section 10.17) for description.

10.16 getSampleStart

getSampleStart(discnum);

Returns the sample start `double` value for the disc referenced. If the value is not known, returns `null`.

10.17 setLength

command	setLength
return type	`void`
syntax	setLength([discnum | `all`], val);

parameters:

[*discnum* | *`all`*] is disc number that this command applies to. `all` is a valid disc reference.
val is the length (in ms) of the sample in sound file established by setFile (section 10.9).

setSampStart and setLength define the part of the sample within the file that will actually be used. Both values are milliseconds.

Example:

If you have a sound file that is 10 seconds long, but only want to use a 1-second portion of that file that starts 2.5 seconds from the start of the file. So setSampStart with val = 2500 and setLength val = 1000; This will give a sample 1 second long starting at 2.5 seconds in the file.

10.18 getLength

getLength(discnum);

Returns the `double` value of the length assigned by setLength. If no value has been set, returns `null`.

10.19 setSlide

command	setSlide
return type	`void`
syntax	setSlide([discnum | `all`], ["up" | "down"], val);
parameters:	

[*discnum* | *`all`*] is the disc number that this command applies to. `all` is a valid disc reference.

[*"up"* | *"down"*] is the string literal which determines if you are addressing acceleration (up) or deceleration (down).

val is the value (`double`) you want to set the slide value to.

slide up and slide down modify the behavior of "seek". Normal seek behavior described in this doc is `true` when both of these values are set to 0.

Higher values effectively smooth out the acceleration and deceleration.

slide up only affects positive (or upward) movements, and slide down only negative/downward.

These setting ONLY affect movement and acceleration/decelerations that are caused by seek commands – slide up/slide down have no affect on speed or acceleration/deceleration caused by setSpeed commands.

With these settings = 0, the result of a seek command is that the disc jumps from a standstill to the speed determined by the position/time values, and when the position is reached, the disc lurches to an instantaneous stop, and the arrival time is precisely what was specified in the seek command.

With these settings > 0, the disc will ramp up gradually to its cruising speed as it seeks towards its destination position, then decelerate gradually as it approaches.

10.20 getSlide

getSlide(discnum [up | down]);

Return the slide value set for the appropriate disc reference. If no value is known, return `null`.

10.21 setSub

command	setSub
return type	`void`
syntax	setSub([send | return], val);

parameters:

[*send* | *return*] are literals used to set appropriate sub feature of Sub. required.

val determines how much of the sound is sent to the subwoofer – 0.0 is none, 1.0 is all – and these are effectively the bounds of this param.

setSub determines how much of the sound for the system is sent to the subwoofer.

10.22 getSub

command	getSub
return type	`double`
syntax	getSub([send | return]);

parameters:

[*send* | *return*] is literal used to set appropriate sub feature of reverb. required.

Returns value set by setSub. `null` if not set in this SPINNER program.

10.23 createDiscs

createDiscs[val];

createDiscs takes a `double` val and truncates fraction part if present. This values indicates the number of discs available in the system. createDiscs returns no value.

10.24 getDiscsSize

getSize();

getDiscsSize returns the number of discs initialized by the system. If the discs have not been initialized, getDiscsSize returns `null`.

10.25 getSize

getSize(arrayreference);

getSize returns the number of elements in the referenced array. If the the reference has not been initialized, getSize returns `null`.

10.26 getTime

getTime returns the absolute time since beginning of simulation as tracked by SPINNER. Time returned in miliseconds.

10.27 Import

import("filename");

Imports a SPINNER file inline and processes. If filename not found, runtime error is thrown. Import statements MUST occur in the first line of any SPINNER program else a compile error will occur.

Chapter 11

Concurrent processing

Most of the discussion in the previous sections dealt with rules for processing a single statement or expression at a time. SPINNER supports the concept of concurrent processing where multiple, concurrent processing paths can execute simultaneously. This section discusses this model.

11.1 SPINscript - Linear execution model

In SPINscript, SPINNER's target language, statements are executed one after another until the end of the file. SPINscript does not support looping, jumping, function calls or parallel execution. This limits the expressive power of the language, and therefore the programmer/artist.

Here is an example of something that is easy to do in SPINscript:

Start discs 1 and 2 spinning at different speeds

Wait for 30 seconds

Change speeds of discs 1 and 2

Here is an example of something that is difficult to do in SPINscript:

Start disc 1 spinning with a speed defined by a sine wave

Start disc 2 spinning with a speed defined by a cosine wave

Wait for 10 seconds

Wait until the speed of disc 1 and disc 2 are nearly equal

Change speeds of disc 1 and 2 to be defined by acceleration function

SPINscript has an execution model which only "enables" two types of speed functions, a linear function via Line and an acceleration function via Speed. It is possible to implement more complicated speed functions in SPINscript, just as it is possible to write object-oriented constructs in C.

It is just not easy. If I wanted to implement the sine and cosine speed functions in SPINscript I would need to manually "interleave" Speed and Wait commands:

```
Speed-1, 8.41, 1000     ; sine of 1 * 10 rev/s for 1 sec
Speed-2, 5.40, 1000     ; cosine of 1 * 10 rev/s for 1 sec
Wait 1000;
Speed-1, 9.09, 1000     ; sine of 2 * 10 rev/s for 1 sec
Speed-2, -4.16, 1000    ; cosine of 2 * 10 rev/s for 1 sec
Wait 1000;
...
```

The programmer has to manually compute the values for the sine and cosine functions at each time step of the simulation. They would also need to manually figure out when the "speed of disc 1 and disc 2 are nearly equal". This is very tedious and error prone which is why it is not done very often.

11.2 SPINNER - Concurrent execution model

SPINNER provides an elegant solution to this problem by supporting concurrent branch execution using the || operator and synchronization using the waitUntil function.

11.2.1 Concurrent branch execution

The concurrent branch operator || allows the user to associate the behavior of the different discs with different parallel blocks of code. Coupled with other nice SPINNER features such as function calls and the ability to query the running simulation properties the solution becomes the following:

```
{
        while(true)
        {
                setSpeed(1, sin(getTime()*10, 1000);
                wait(1000);

                if (getSpeed(2) - getSpeed(1) < 5)
                {
                        break;
                }
        }
} || {
        while(true)
        {
                setSpeed(2, cos(getTime()*10, 1000);
                wait(1000);
                if (getSpeed(2) - getSpeed(1) < 5)
                {
                        break;
                }
        }
}

setSpeed(1, 1000);
setSpeed(2, 1000);
```

In this example, two branches of execution are running in parallel (actually, “lockstep” is the more accurate term). The first branch updates the speed of disc 1 every second based on the sine function. The second one updates disc 2 based on cosine. When their speeds are nearly equal the branches terminate and execution resumes in the main branch.

11.2.2 Comparison of concurrent branch execution (CBE) and threading

Threads and CBE are very similar in that their execution is “interleaved” and the ordering is dependent on the scheduling algorithm. For SPINNER, the scheduling algorithm is round-robin with a 1 instruction “quanta”. This means that each branch executes one instruction in a round-robin fashion, from the first branch to the last.

In CBE the parent branch blocks waiting for the child branches to terminate. However, with threads the parent thread continues execution and is scheduled just like the child threads.

The dependency of the parent branch on the child means that the programmer must be careful not to allow a branch to block indefinitely. This is not a problem for threads and, in fact, its very useful for threads to be blocked waiting for input, I/O or an event. This flexibility and power come at a price as multi-threaded programming is notoriously difficult to “get right”.

As a programming construct, CBE is declarative whereas multi-threading is dynamic. That is, threading requires the user to allocate and manage thread data structures. SPINNER, on the other hand, implicitly manages the concurrently executing branches. As mentioned before, this simplifies the programming model but constrains the functionality due to the parent-child dependency of the branches.

11.2.3 Synchronization

Often the artist wants to synchronize the behavior of two or more discs. For instance, the artist may want to wait until the angle of disc 1 and 2 are within some amount of each other and then synchronize the speed of disc 1 with that of disc 2. Because SPINscript does not provide any support for this, the artist had to manually calculate how long it would take.

SPINNER provides better support for this functionality with the waitUntil function. With this function, the artist can represent the synchronization condition as a Boolean expression:

```
{
        setSpeed(1, 10, 1000);
        waitUntil(getAngle(2) - getAngle(1) < 5);
        setSpeed(1, getSpeed(2));
        wait(5000);
} || {
        seek(2, 10, 5000);
        wait(5000);
}
```

Chapter 12

References

As SPINNER is implemented in Java, many areas of this reference manual were taken nearly *verbatim* from *The Java Language Specification - Second Edition* by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.

The C Programming Language, second edition by Brain Kernighhan and Dennis Ritchie was used for many of the lexical conventions of SPINNER.

If something here looks like it exists in the JLS or C LRM, it is safe to assume it is taken from there. (The sincerest form of flattery.)

Appendix A

The Grammar

```
main_statement
        :outer_statement
        |func_definition
;

outer_statement
        :statement
        |declaration SEMI
        |parallel_statement
;

statement
        :if_statement
        |for_statement
        |while_statement
        |repeat_statement
        |assignment SEMI!
        |func_call SEMI!
        |create_disk SEMI!
        |SEMI!
;

statement_for_func
        :statement
        |return_stmt SEMI
        |declaration SEMI
        |parallel_statement
;

return_stmt
        : "return" (expr)?
;

inner_statement
        :statement
        |break_statement
        |LBRACE! (main_statement)* RBRACE!
;

break_statement
        : "break" SEMI!
;

if_statement
        : "if" LPAREN! expr RPAREN! "then"! inner_statement
        (options {greedy=true;} : "else"! inner_statement)?
;

for_statement
        : "for" LPAREN! (for_init)? SEMI! (expr)? SEMI! (for_increment)? RPAREN! inner_stateme
;

for_init
        :assignment
        |declaration
;

for_increment
        :assignment
;

while_statement
        : "while" LPAREN! expr RPAREN! "do" inner_statement
;

repeat_statement
```

```
        : "repeat" inner_statement "until" LPAREN! expr RPAREN!
;

parallel_statement
        : LBRACE! (outer_statement)* RBRACE! (PARALLEL! LBRACE! (outer_statement)* RBRACE!)+
;

create_disk
        : "createDisc"! LBRAKET! NUMBER RBRAKET
;

declaration
        : variable_declaration
;

variable_declaration
        : type (ID | assignment_with_decl | array_decl)
;

type
        : "double"
        | "boolean"
        | "void"
;

func_definition
        : type "func" ID LPAREN! (args)? RPAREN! LBRACE! func_body RBRACE!
;

func_body
        : (statement_for_func)*
;

args
        : arg (COMMA! arg)*
;

arg
        : type ID
;
```

```
func_call
        : ID LPAREN (params)? RPAREN
;

params
        : expr (COMMA! expr)*
;

assignment_with_decl
        :ID ASSIGN! expr
        |auto_inc_assgn
;

assignment
        :assignment_with_decl
        |array ASSIGN expr
;

auto_inc_assgn
        :ID INC
        |ID DEC
        |INC ID
        |DEC ID
;

expr
        :logic_expr1 ( "or" logic_expr1)*
;

logic_expr1
        :logic_expr2 ("and" logic_expr2)*
;

logic_expr2
        :("not")? relational_expr
;

relational_expr
        :arithmatic_expr1 ((GT | LT | GE | LE | EQ | NE ) arithmatic_expr1)?
```

```
;

arithmatic_expr1
        :arithmatic_expr2 ((PLUS | MINUS) arithmatic_expr2)*
;

arithmatic_expr2
        :arithmatic_expr3 ((MULT | DIV | MOD) arithmatic_expr3)*
;

arithmatic_expr3
        : (PLUS | MINUS)? rvalue
;

rvalue
        : func_call
        | NUMBER
        | "true"
        | "false"
        | LPAREN! expr RPAREN!
        | "all"
        | "null"
        | ID
        | STRING
        | array
;

array
        : ID LBRAKET! expr RBRAKET!
;

array_decl
        : array (ASSIGN! LBRACE! expr (COMMA! expr)* RBRACE!)?

;
```

Appendix B

Function library - work in progess

B.1 Rtz([disc | `all`]);

B.2 getAngle(discnum);

B.3 seekMany([discnum | `all`], array, array)

B.4 cosine

B.5 sine

Appendix C

Example Programs

C.1 Program 1

```
/*SPINNER Sample program 1
  author:Gaurav
*/

//initializations

setGain(all,.25);
setReverb(send,1.0);
if (not(getReverb(send) = null)) then {
        setSub(send,1.0);
} else {;}

//empty statement
;

//discs creation and setup
//global variable
createDisc[5];

if (getGain(1)=.25) then {
        setFile(1,"mixer.wav");
}

//set the sound file for all other discs as "melody.wav"
```

```
for (double i:=2;i<=num;i++) {
        setFile(i,"melody.wav");
}

if (isFileSet(1)) then {
        setSpinMult(1,.5);
}
if (getSampleStart(1) = null) then {
        setSampleStart(1,3000);
        setLength(1,2000);
}

setSlide(all,up,5000);
setSlide(all,down,5000);

//disc setup complete

//importing functions
import ("disclib1.spin");

/*a small simulation in which all the discs make 5 revolutions
at 1 rev/sec clockwise and then 5 revolutions at 2 rev/sec ccw
we call this simulation as a standard simulation and make it a function
*/

void func standardSimul() {
        seek(all,5,5000);
        wait(5000);
        seek(all,0,2500);
        wait(2500);
        return;
}

//if all discs are at position 0 then run the standard simulation

boolean flag := true;
for (double i:=1;i<=num;i++) {
        if(not(getPostion(i)=0)) then {
                flag:=false;
        }
}

if(flag) then {
        standardSimul();
}

Rtz(all); // a Lib function

//Done with this program. do quick clean-up
cleanup();  //found in disclib1
```

C.2 Program 2

```
/*SPINNER Sample program 2
  author:Gaurav
*/

import("disclib1.spin");

//initializations
setGain("master",.5);
setReverb("return",.5);
setGain(.5);

//discs setup
createDisc[3];
setFile(all,"remix.wav");
setSampleStart(1,1000);
setSampleStart(2,2000);
setSampleStart(3,3000);
setLength(all,2000);
setSpinMult(1,1.0);
setSpinMult(2,2.0);
setSpinMult(3,-2.0);
setSlide(all,up,0);
setSlide(all,down,0);

/*fork processes.
```

```
we start of disc 1 at 1 rev/sec and disc 2 at 2 rev/sec clockwise
simultaneously and when when disc 2's postion > 5 we set disc 1's speed = disc 2
*/

{
        seek(1,10,10000);
        waitUntil(getPosition(2) > 5,-1);
        setSpeed(1,getSpeed(2),0);
}
||
{
        seek(2,10,5000);
        wait(5000);
}

Rtz(all);
/*In this function disc x starts rotating clockwie at 1 rev/sec and dics y at 2 rev/sec clockwi
*/
void func simulCall(double x, double y) {
        boolean flag:=false;
        {
                seek(x,10,10000);
                waitUntil(flag,10000);
                setSpeed(x,y,0);
        }
        ||
        {
                seek(y,10,5000);
                wait(2000);
                flag:=true;
                wait(2000);
                setSpeed(y,x,0);
        }
}

//nested if's, concurrent processing and function call;
if (getPostion(1)=0 and getPosition(2)=0) then {
        if(getSpeed(3)=0 or getPostion(3)=0) then {
                {
                        simulCall(1,2);
                }
                ||
                {
                        setSpeed(3,1.0,1.0);
                }
        }
}

Rtz(all);
cleanup();
```